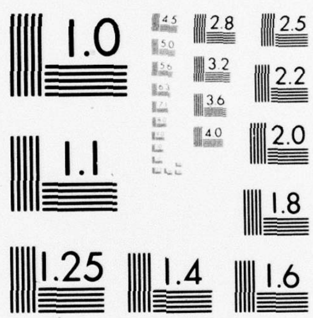
AD-A039 747 FEDERAL COBOL COMPILER TESTING SERVICE WASHINGTON D C F/G 9/2
THE DOD COBOL COMPILER VALIDATION SYSTEM, (U)
MAY 77 G N BAIRD
UNCLASSIFIED FCCTS/TR-77/03 NL

1 OF 1
ADA039 747

END
DATE
FILMED
6-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

The DOD COBOL compiler validation system

by GEORGE N. BAIRD

Department of the Navy
Washington, D. C.

INTRODUCTION

The ability to benchmark or validate software to ensure that design specifications are satisfied is an extremely difficult task. Test data, generally designed by the creators of said software, is generally biased toward a specific goal and tend not to cover many of the possibilities of combinations and interactions. The philosophy of suggesting that "a programmer will never do . . ." or "this particular situation will never happen" is altogether absurd. First, "never" is an extremely long time and secondly, the Hagel theorem of programming states that "if it can be done, whether absurd or not, one or more programmers will more than likely try it."

Therefore, if a particular piece of software has been thoroughly checked against all known extremes and a majority of all syntactical forms, then the Hagel theorem of programming will not affect the software in question. The DOD CCVS attempts to do just that by checking for the fringes of the specifications of X3.23-1968[1] and known limits. It is assumed that a COBOL compiler will perform satisfactorily for the audit routines, then it is likely that the compiler supports the entire language. However, if the computer has trouble with handling the routines in the CCVS it can be assumed that there will indeed be other errors of a more serious nature.

The following is a brief account of the history of the DOD CCVS, the automation of the system and the adaptability of the system to given compilers.

BACKGROUND

The first revision to the initial specification for COBOL (designated as COBOL-1961[2]) was approved by the Executive Committee of the Conference on Data Systems Languages* and published in May of 1961. Recognizing that the language would be subject to additional development and change, an attempt was made to create uniformity and predictability in the various implementations of COBOL compilers. The language elements were placed in one of two categories: required and elective.

Required COBOL-1961 consisted of language elements (features and options) which must be implemented by any implementor claiming a COBOL-1961 compiler. This established a common minimum subset of language elements for COBOL compilers and hopefully a high degree of transferability of source programs between compilers if this subset was adhered to.

Elective COBOL-1961 consisted of language elements whose implementation had been designated as optional. It was suggested that if an implementor chose to include any of these features (either totally or partially) he would be expected to implement these in accordance with the specifications available in COBOL-1961. This was to provide a logical growth for the language and attempt to prevent a language element from having contradictory meaning between the language development specifications and implementor's definition.

As implementors began providing COBOL compilers based on the 1961 specifications, unexpected problems became somewhat obvious. The first problem was that the specifications themselves suggested mandatory as well as optional language elements for implementing COBOL compilers. In addition the development docu-

* The Conference on Data Systems Languages (CODASYL) is an informal and voluntary organization of interested individuals supported by their institutions who contribute their efforts and expenses toward the ends of designing and developing techniques and languages to assist in data systems analysis, design, and implementation. CODASYL is responsible for the development and maintenance of COBOL.

ment produced by CODASYL was likely to change periodically thus, providing multiple specifications to implement from. Compilers could consist of what the implementor chose to implement which would severely handicap any chance of transferability of programs among the different compilers, particularly since no two implementors necessarily think alike. Philosophies vary both in the selection of elements for a COBOL compiler and in the techniques of implementing the compiler itself. (As ridiculous as it may sound, some compilers actually scan, syntax check and issue diagnostics for COBOL words that might appear in comments both in the REMARKS paragraph of the Identification Division and in NOTE sentences in the Procedure Division.) The need for a common base from which to implement became obvious. If the language was to provide a high degree of compatability, then all implementations had to be based on the same specification.

The second problem was the reliability of the compiler itself. If the manual for the compiler indicated that it supported the DIVIDE statement, the user assumed this was true. If the compiler then accepted the syntax of the DIVIDE statement, the user assumed that the object code necessary to perform the operation was generated. When the program executed, he expected the results to reflect the action represented in his source code. It appears that in some cases perhaps no code was generated for the DIVIDE statement and the object program executed perfectly except for the fact that no division took place. In another case, when the object program encountered the DIVIDE operation, it simply went into a loop or aborted. At this point, the programmer could become decidedly frustrated. The source code in his program indicated that: (1) he requested that a divide take place, (2) there was no error loop in his program, (3) the program should not abort. This is the problem we are addressing: A programmer should concern himself with producing a source program that is correct logically and the necessary operating system control statements to invoke the COBOL compiler. In doing so, he should be able to depend on the compiler being capable of contributing its talent in producing a correct object program.

If the user was assured that either: (1) each instruction in the COBOL language had been implemented correctly, or, (2) that each statement which was implemented did not give extraneous results, then the above situation could not exist.

Thus, the need for a validation tool becomes apparent. Although all vendors exercise some form of quality control on their software before it is released, it is clear that some problems may not be detected. (The initial release of the Navy COBOL audit routines revealed over 50 bugs in one particular compiler which had been released five years earlier.)

By providing the common base from which to implement and a mechanism for determining the accuracy and correctness of a compiler relative to the specification, the problem of smorgasbord compilers (that may or may not produce expected results) should become extinct.

The standardization of COBOL began on 15 January 1963. This was the first meeting of the American Standards Association Committee, X3.4.4,* the Task Group for Processor Documentation and COBOL. The program of work for X3.4.4 included . . . "Write test problems to test specific features and combinations of features of COBOL. Checkout and run the test problems on various COBOL compilers." A working group (X3.4.4.2) was established for creating the "test problems" to be used for determining feature availability.

The concept of a mechanism for measuring the compliance of a COBOL compiler to the proposed standard seemed reasonable in view of the fact that other national standards did indeed lend themselves to some form of verifications, i.e., 2×4's, typewriter keyboards, screw threads.

IMPLEMENTING A VALIDATION SYSTEM FOR COBOL

In order to implement a COBOL program on a given system, regardless of whether the program is a validation routine or an application program, the following must be accomplished:

1. The special characters used in COBOL (i.e., '(', ')', '*', '+', '<' etc.) must be converted for the system being utilized.†
2. All references to implementor-names within each of the source programs must be resolved.
3. Operating System Control Cards must be pro-

* The American Standards Association (ASA), a voluntary national standards body evolved to the United States of America Standards Institute (USASI) and finally the American National Standards Institute (ANSI). The committee X3.4.4 eventually became X3J4 under a reorganization of the X3 structure. X3J4 is currently in the process of producing a revision to X3.23-1968.

† For most computers the representatives for the characters A-Z, 0-9, and the space (blank character) are the same. However, there is sometimes a difference in representation of the other characters and therefore conversion of these characters from one computer to another may be necessary.

duced which will cause each of the source programs to be compiled and executed. Additionally, the user must have the ability to make changes to the source programs, i.e., delete statements, replace statements, and add statements.

4. As the programs are compiled, any statements that are not syntactically acceptable to the compiler must be modified or "deleted" so that a clean compilation takes place and an executable object program is produced.

5. The programs are then executed. All execution time aborts must be resolved by determining what caused the abort and after deleting or modifying that particular test or COBOL element, repeating steps 3 and 4 until a normal end of job situation exists.

Development of audit routines

March 1963, X3.4.4.2 (the Compiler Feature Availability Working Group) began its effort to create the COBOL programs which would be used to determine the degree of conformance of a compiler to the proposed standard. The intent of the committee was not to furnish a means for debugging compilers, but rather to determine "feature availability." Feature availability was understood to mean that the compiler accepted the syntax and produced object code to produce the desired result. All combinations of features were not to be tested; only a carefully selected sample of features (singly and in combination) were to be tested to insure that they were operational. The test programs themselves were to produce a printed report that would reflect the test number and when possible whether the test "Passed" or "Failed." See Figure 1.

When a failure was detected on the report, the user could trace the failure to the source code and attempt

```
Source Statements
TEST-0001.
    MOVE 001 TO TEST-NO.
    MOVE ZERO TO ALPHA.
    ADD 1 TO ALPHA.
    IF ALPHA = 1 PERFORM PASS ELSE PERFORM FAIL.
TEST-0002.
Results
    TEST    NO    P - F
    ADD      1    P
      .
      .
    ADD     21        F
```

Figure 1—Example of X3.4.4.2 test and printed results

to identify the problem. The supporting code (printing routine, pass routine, fail routine, etc.) was to be written using the most elementary statements in the low-level of COBOL. The reason for this was twofold:

1. The programs would be able to perform on a minimum COBOL compiler (Nucleus level 1, Table Handling level 1, and Sequential Access level 1).
2. The chances of the supporting code not being acceptable to the compiler being tested were lessened.

The programs, when ready, would be provided in card deck form along with the necessary documentation for running them. (The basic philosophies of design set forth by X3.4.4.2 were carried through all subsequent attempts to create compiler validation systems for COBOL.)

Assignments were made to the members of the committee and the work began. This type of effort at the committee level, however, was not as productive as the work of standardizing the language itself.

In April 1967, the Air Force issued a contract for a system to be designed and implemented which could be used in measuring a compiler against the standard. The Air Force COBOL Compiler Validation System was to create test programs and adapt them to a given system automatically by means of fifty-two parameter cards.

The Navy COBOL audit routines

In August of 1967, The Special Assistant to the Secretary of the Navy created a task group to influence the use of COBOL throughout the Navy. Being aware of both the X3.4.4.2 and Air Force efforts, (as well as the time involved for completion), a short term project was established to determine the feasibility of validating COBOL compilers. After examining the information and test programs available at that time, the first set of routines was produced. In addition to the original X3.4.4.2 philosophy, the Navy added the capability of providing the result created by the computer as well as the expected result when a test failed. Also, instead of a test number, the actual procedure name in the source program was reflected in the output. See Figure 2.

The preliminary version of the Navy COBOL audit routines was made up of 12 programs consisting of about 5000 lines of source code. The tailoring of the programs to a particular compiler was done by hand (by physically changing cards in the deck or by using the vendor's software for updating COBOL programs). As tests were deleted or modified, it was difficult to bring the programs back to their virgin state for subsequent runs against different compilers or for determining what changes had to be made in order that the programs would execute.

This was a crude effort, but it established the necessary evidence that the project was feasible to continue and defined techniques for developing auditing systems. Because of the favorable comments received on this initial work done by the Navy, it appeared in the best interest of all to continue the effort.

After steady development and testing for a year, Version 4 of the Navy COBOL Audit Routines was released in December 1969. The routines consisted of 55 Programs, consisting of 18,000 card images capable of testing the full standard. The routines had also become one of the benchmarks for all systems procured by the Department of the Navy in order to ensure that the compiler delivered with the system supported the required level of American National Standard COBOL.*

Also, Version 4 introduced the VP-Routine, a program that automated the audit routines. Based on fifty parameter cards, all implementor-names could be resolved and the test programs generated in a one-pass operation. See Figure 3.

In addition, by coding specific control cards in the Working-Storage Section of the VP-Routine as constants, the output of the VP-Routine became a file that very much resembled the input from a card reader, i.e., control cards, programs, etc.

By specifying the required Department of Defense COBOL subset of the audit routines to be used in a validation, only the programs necessary for validating

Source Statements

```
ADD-TEST-1.
  MOVE 1 TO ALPHA.
  ADD 1 TO ALPHA.
  IF ALPHA =2 PERFORM PASS ELSE PERFORM FAIL.
  .
  .
  .
```

Results

FEATURE	PARAGRAPH	P/F	COMPUTED	EXPECTED
ADD	ADD-TEST-1	FAIL	1	2
ADD	ADD-TEST-2	PASS		

Figure 2—Example of Navy test and printed results

* In 1968, the Department of Defense, realizing that several thousand combinations of modules/levels were possible, established four subsets of American National Standard COBOL for procurement purposes.

V-P Routine Input:

```
X-0  SOURCE-COMPUTER-NAME
X-1  OBJECT-COMPUTER-NAME
X-3
  .
  .
X-8  PRINTER
X-9  CARD-READER
X-10
  .
  .
  .
X-50
```

Audit Routine File:

```
SOURCE-COMPUTER.
     XXXXX0
       .
       .
     SELECT PRINT-FILE ASSIGN TO
     XXXXX8
```

The audit routine after processing would be:

```
SOURCE-COMPUTER.
     SOURCE-COMPUTER-NAME.
       .
       .
     SELECT PRINT-FILE ASSIGN TO
     PRINTER.
```

Figure 3—Example of input to the support routine, Population file where audit routines are stored and resolved audit routine after processing

that subset of elements or modules would be selected, i.e., SUBSET-A, B, C, or D. The capability also existed to update the programs as the "card reader" file was being created. The use of the VP-Routine was not mandatory at this time, but merely to assist the person validating the compiler in setting up the programs for compilation. Once the VP-Routine was set up for a given system, there was little trouble running the audit routines. The user then had only to concern himself with the validation itself and with achieving successful results from execution of the audit routines. When an updated set of routines was distributed, there was no effort involved in replacing the old input tape to the VP-Routine with the new tape.

The Air Force COBOL audit routines

The Air Force COBOL Compiler Validation System (AFCCVS) was not a series of COBOL programs but rather a test program generator. The user could select

```
Source statement in test library
T 1N078A101NUC, 2NUC                                              4U
400151  77  WRK-DS-18V00                        PICTURE  S9(18).
400461  77  A180NES-DS-18VOO                    PICTURE  S9(18).
400471                                        VALUE 1111111111111111.
400881  77  A18ONES-CS-18VOO                    PICTURE S9(18) COMPUTATIONAL
400891                                        VALUE 111111111111111111.
802925  TEST-1NUC-078.
802930      MOVE A18ONES-DS-18VOO                          TO WRK-DS-18VOO.
802935      ADD A180NES-CS-18VOO                           TO WRK-DS 18VOO
802940      MOVE WRK-DS-18VOO                              TO SUP-WK-A.
802945      MOVE '222222222222222222'                      TO SUP-WK-C.
802950      MOVE '1NO78'                                   TO SUP-ID-WK-A
802955      PERFORM SUPPORT-RTN THRU SUP-TRN-C.
Test results
  .1NO78              .1NO79.
  .222222222222222222.09900.
  .                   .——.
```

Figure 4—Example of Air Force test and printed results

the specific tests or modules he was interested in and the AFCCVS would create one or more programs from a file of specific tests which were then compiled as audit routines. Implementor-names were resolved as the programs were generated based on parameter cards stored on the test file or provided by the user.

The process required several passes, including the sorting of all of the selected tests to force the Data Division entries into the Data Division and place the tests themselves in the Procedure Division where they logically belonged. An additional pass was required to eliminate duplicate Data Division entries (more than one test might use the same data-item and therefore there would be more than one copy in the Data Division). See Figure 4.

Still another program was used to make changes to the source programs as the compiler was validated. As in the Navy system, certain elements had to be eliminated because: (1) they were not syntactically acceptable to the compiler or, (2) they caused run time aborts.

Department of Defense COBOL validation system

In December 1970, The Deputy Comptroller of ADP in the Office of the Secretary of Defense asked the Navy to create what is now the DOD Compiler Validation System for COBOL taking advantage of: (1) the better features of both the Navy COBOL Audit Routines (Version 4) and the Air Force CCVS and (2) the four years of in-house experience in designing and implementing audit routines on various systems as well as the actual validation of compilers for procurement purposes.

The Compiler Validation System (of which the support program was written in COBOL) had to be readily adaptable to any computer system which supported a COBOL compiler and which was likely to be bid on any RFP issued by the Department of Defense or any of its agencies. It also had to be able to communicate with the operating system of the computer in order to provide an automated approach to validating the COBOL compiler. The problem of interfacing with an operating system may or may not be readily apparent depending on whether an individual is more familiar with IBM's Full Operation System (OS), which is probably the most complex operating system insofar as establishing communication between itself and the user is concerned, or with the Burroughs Master Control Program (MCP), where the control language can be learned in a fifteen or twenty minute discussion.

Since validating a compiler may not be necessary very often, the amount of expertise necessary for communicating with the CVS should be kept to a minimum. The output of the routines should be as clear as possible in order not to confuse the reviewer of the results or to suggest ambiguities.

The decision was made to adopt the Navy support system and presentation format for several reasons. (1) It would be easier to introduce the Air Force tests into the Navy routines as additional tests because the Navy routines were already in COBOL program format. It would have been difficult to recode each of the Navy tests into the format of specific tests on the Air Force Population File because of the greater volume of tests. (2) The Navy support program had become rather versatile in handling control cards, even for IBM's OS, whereas the Air Force system had only limited control card generation capability.

The merging of the Air Force and Navy routines

The actual merging of the routines started in February 1971 and continued until September 1971. During the merging operation, it was noted that there was very little overlap or redundancy in the functions tested by the Air Force and Navy systems. In actuality, the two sets of tests complemented each other. This could only be attributed to the different philosophies of the two organizations which originally created the routines. For example in the tests for the ADD statement:

Air Force	*Navy*
signed fields	unsighed fields
most fields 18 digits long	most fields 1-10 digits long
more computational items	more display items

After examining the Add tests for the combined DOD routines, it was noticed that a few areas had been totally overlooked.

1. An ADD statement that forced the "temp" used by the compiler to hold a number greater than 18 digits in length:

   ```
   i.e., ADD  +999999999999999999
              +999999999999999999
              +999999999999999999
              -999999999999999999
              -999999999999999999
              -99  TO ALPHA
   ```

   . . . where the intermediate result would be greater than 18 digits, but the final result would be able to fit in the receiving field.
2. There were not more than eight operands in any one ADD test.
3. A size error test using a COMPUTATIONAL field when the actual value could be greater than the described size of the field, i.e., ALPHA PICTURE 9(4) COMP. . . specifies a data item that could contain a maximum value of 9999 without an overflow condition; however, because the field may be set up internally in binary, the decimal value may be less than the maximum binary value it could hold:

   Maximum COBOL value = 9999
   Maximum hardware value ≃ 16383

Therefore, from this point of view, the merging of the routines disclosed the holes in the validation systems being used prior to the current DOD routines.

Source statements

```
ADD-TEST-1.
    MOVE 1 TO ALPHA.                  Initialization if
                                        necessary.
    ADD 1 TO ALPHA.                   The Test.
    IF ALPHA = 2                      Check the results of the
        PERFORM PASS                    test and handle the
                                        accounting of that
                                        test.
     ELSE
        GO TO ADD-FAIL-1.
    GO TO ADD-WRITE-1.                Normal exit path to the
                                        write paragraph.
ADD-DELETE-1.                         Abnormal path to the
    PERFORM DELETE.                     write statement if the
    GO TO ADD-WRITE-1.                  test is deleted via the
                                        NOTE statement.
ADD-FAIL-1.                           Correct and computed
    MOVE ALPHA TO COMPUTED.             results are formatted
    MOVE '2' TO CORRECT.                for printing.
    PERFORM FAIL.
ADD-WRITE-1                           Results are printed.
    MOVE 'ADD-TEST-1' TO PARAGRAPH-NAME.
    PERFORM PINT-RESULTS.
ADD-TEST-2.
```

Figure 5—Example of DOD test and supporting code

The general format of each test is made up of several paragraphs: (1) the actual "test" paragraph; (2) a "delete" paragraph which takes advantage of the COBOL NOTE for deleting tests which the compiler being validated cannot handle; (3) the "fail" paragraph for putting out the computed and correct results when a test fails; and (4) a "write" paragraph which places the test name in the output line and causes it to be written. See Figure 5.

The magnitude of the size of the DOD Audit Routines was approaching 100,000 lines of source coding, making up 130 programs. The number of environmental changes (resolution of implementor-names) was in the neighborhood of 1,000 and the number of operating system control cards required to execute the program would be from 1,300 to 5,000 depending on the complexity of the operating system involved.

This was where the support program could save a large amount of both work and mistakes. The Versatile Program Management System (VPMS1) was designed to handle all of these problems with a minimum of effort.

Versatile program management system (VPMS1)

A good portion of the merging included additional enhancements to the VPMS1 (support program) which, by this time, through an evolutionary process had learned to manage two new languages; FORTRAN and JOVIAL. The program had been modified based on the additional requirements of various operating systems for handling particular COBOL problems; the need for making the system easy for the user to interface with, and the need to provide all interfaces between the user, the audit routines, and the operating system.

The introduction of implementor names through "X-cards"

The first problem was the resolution of implementor-names within the source COBOL programs making up the audit routines. In the COBOL language, particularly in the Environment Division, there are constructs which must contain an implementor-defined word in order for the statement to be syntactically complete. Figure 6 shows where the implementor-names must be provided.

THE NOTE placed as the first word in the paragraph causes the entire paragraph to be treated as comments. Instead of the "GO TO ADD-WRITE-1" statement being executed, the logic of the program falls into the delete paragraph which causes the output results to reflect the fact that the test was deleted.

If the syntax error is in the Data Division, then the coding itself must be modified. VPMS1 shows, in its own printed output, the old card image as well as the new card image so that what has been altered is readily apparent, i.e.,

```
012900 02 A PIC ZZ9 Value '1'. NC1085.2 OLD
012900 02 A PIC ZZ9 Value 1.   NC108*RE NEW
```

```
ENVIRONMENT DIVISION.
SOURCE-COMPUTER.
    implementor-name-1.
OBJECT-COMPUTER.
    implementor-name-2.
SPECIAL-NAMES.
    implementor-name-3 is MNEMONIC-NAME
              .
              .
              .
FILE-CONTROL
    SELECT FILE-NAME ASSIGN TO implementor-name-4.
              .
              .
              .
data division.
    FD FILE-NAME
        VALUE OF implementor-name-5 IS implementor-defined.
```

Figure 6—Implementor defined names that would appear in a COBOL program

If, while executing the object program of an audit routine, an abnormal termination occurs, then a change is required. The cause might be, for example, a data exception or a program loop due to the incorrect implementation of a COBOL statement. In any case, the test in question would have to be deleted. The NOTE would be used as specified above.

In addition, VPMS1 provides a universal method of updating source programs so that the individual who validates more than one compiler is not constantly required to learn new implementor techniques for updating source programs.

Example of update cards through VPMS1:

```
012900  02 A PIC ZZ9     (If the sequence number is
            VALUE 1.      equal the card is replaced;
013210  MOVE 1 TO A.      if there is no match the
014310  NOTE              card is inserted in the ap-
                          propriate place in the
                          program.)
014900*            (Deletes card 014900)
029300*099000      (Deletes the series from 029300
                     through 099000).
```

To carry the problem a step further. Some of the names used by different implementors for the high speed printer in the SELECT statement have been PRINTER, SYSTEM-PRINTER, FORM-PRINTER, SYSOUT, SYSOU1, P1 FOR LISTING, ETC. It is obvious to a programmer what the implementor has in mind, but the compiler that expects SYSTEM-PRINTER, will certainly reject any of the other names. Therefore, each occurrence of an implementor-name must be converted to the correct name. The approach taken is that each implementor-name is defined to VPMS1. For example, the printer is known as XXXX36 and the audit routines using the printer would be set up in the following way:

```
SELECT PRINT-FILE ASSIGN TO
  XXXXX36
```

And the user would provide the name to be used by the computer being tested through an "X-CARD."

```
X-36 SYSTEM-PRINTER
```

VPMS1 would then replace all references of XXXXX36 with SYSTEM-PRINTER.

```
SELECT PRINT-FILE ASSIGN TO
SYSTEM-PRINTER.
```

Ability to update programs

The next problem was to provide the user with a method for making changes to the audit routines in

```
ADD-TEST-1.
    NOTE (Inserted by the user as an update to the program.)
    MOVE 1 TO ALPHA.

    TO TO ADD-WRITE-1.
ADD-DELETE-1.
    PERFORM DELETE.
```

Figure 7—Example of deleting a test in the DOD CCVS

an orderly fashion and at the same time provide a maximum amount of documentation for each change made. There are two reasons for the user to need to make modifications to the actual audit routines:

a. If the compiler will not accept a form of syntax it must be eliminated in order to create a syntactically correct program. There are two ways to accomplish this. In the Procedure Division the NOTE statement is used to force the "invalid" statements to become comments. The results of this action would cause the test to be deleted and this would be reflected in the output. See Figure 7.

OPERATING SYSTEM CONTROL CARD GENERATION

The third problem was the generation of operating system control cards in the appropriate position relative to the source programs in order for the programs to be compiled, loaded and executed. This was the biggest challenge for VPMS1; a COBOL program which had to be structurally compatible with all COBOL compilers and which also had to be able to interface with all operating systems with a negligible amount of modification for each system.

The philosophy of the output of VPMS1 is a file acceptable to a particular operating system as input. For the most part this file closely resembles what would normally be introduced to the operating system through the system's input device or card reader, i.e., control cards, source program, data, etc.

The generation of operating system control cards is based on the specific placement of the statement and the requirement or need for specific statements to accomplish additional functions. These control cards are presented to VPMS1 in a form that will not be intercepted by the operating system and are annotated as to their appropriate characteristics. The body of the actual control card starts in position 8 of the input record. Position one is reserved for a code that specifies the type of control card. The following is allowed in specifying control cards: Initial control cards are generated once at the beginning of the file. Beginning control cards are generated before each source program with a provision for specifying control cards which are generated at specific times, i.e., JOB type cards, subroutine type cards, library control cards, etc. Ending control cards are generated after each source program with the same provision as beginning control cards. Terminal control cards are generated prior to the file being closed. Additional control cards are generated for assigning hardware devices to the object program, bracketing data and for assigning work areas to be used by the COBOL Sort.

There are approximately 25 files used by the entire set of validation routines for which control cards may need to be prepared. In addition to the control cards and information for the Environment Division, the total number of control statements printed for VPMS1 could be in the neighborhood of 200 card images and the possible number of generated control cards on the output file could be as large as 5000. The saving in time and JCL errors that could be prevented should be obvious at this point.

This Environmental information need not be provided by the user because once a set of VPMS1 control cards has been satisfactorily debugged on the system in question, they can be placed in the library file that contains the same program so that a single request could extract the VPMS1 control cards for a given system.

CONCLUSION

It has been demonstrated that the validation of COBOL compilers is possible and that the end result is beneficial to both compiler writers and the users of these compilers. The ease with which the DOD CCVS can be automatically adapted to a given computer system has eliminated approximately 85 to 90 percent of the work involved in validating a COBOL compiler.

Although most compilers are written from the same basic specifications (i.e., the American National Standard COBOL, X3.23-1968, or the CODASYL COBOL Journal of Development) the results are not always the same. The DOD CCVS has exposed numerous compiler bugs as well as misinterpretations of the language. Due to this and similar efforts in the area of compiler validation, the compatibility of today's compilers has grown to a high degree.

We are now awaiting the next version of the American National Standard COBOL. The new specifications will provide an increased level of compatibility between compilers because the specifications are more definitive and contain fewer "implementor defined" areas. In addition, numerous enhancements and several clarifications have been included in the new specification—all contributing to better software, both at the compiler and the application level.

REFERENCES

1 American National Standard COBOL X3.23-1968
American National Standards Institute Inc. New York 1968
2 COBOL-61 Conference on Data System Languages
U. S. Government Printing Office Washington D. C. 1961

14

BIBLIOGRAPHIC DATA SHEET	1. Report No. FCCTS/TR-77/03	2.	3. Recipient's Accession No.

6

4. Title and Subtitle
The DOD COBOL Compiler Validation System

11

5. Report Date
9 May 1977

6.

10

7. Author(s)
George N. Baird

8. Performing Organization Rept. No.

9. Performing Organization Name and Address
Federal COBOL Compiler Testing Service
ADPE Selection Office
Department of the Navy
Washington, D. C. 20376

12 10p.

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address
ADPE Selection Office
Department of the Navy
Washington, D. C. 20376

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

This technical report goes into the history of COBOL compiler validations. It describes several early efforts by various committees and of different agencies within the Federal Government. The main thrust of the report is the COBOL Compiler Validation System which was produced by the Department of the Navy after participating in, and reviewing the, results of the less than successful efforts that preceded the work of the Navy. It discusses the implementation problem surrounding the CCVS (which would be applicable to any COBOL system) and the automated source program libraries (VP-routine) developed to speed up the implementation by resolving implementor-names, providing a technique for updating the source program, and generating the necessary Job Control Language necessary to compile and execute each of the audit routines making up the CCVS.

17. Key Words and Document Analysis. 17a. Descriptors

COBOL
Validation
Software
Audit Routines
Verifying
Compilers
Standards
Programming Languages

17b. Identifiers/Open-Ended Terms

CCVS

17c. COSATI Field/Group 09/02

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release unlimited.	UNCLASSIFIED	9
	20. Security Class (This Page) UNCLASSIFIED	22. Price